Carnegie Mellon
Software Engineering Institute

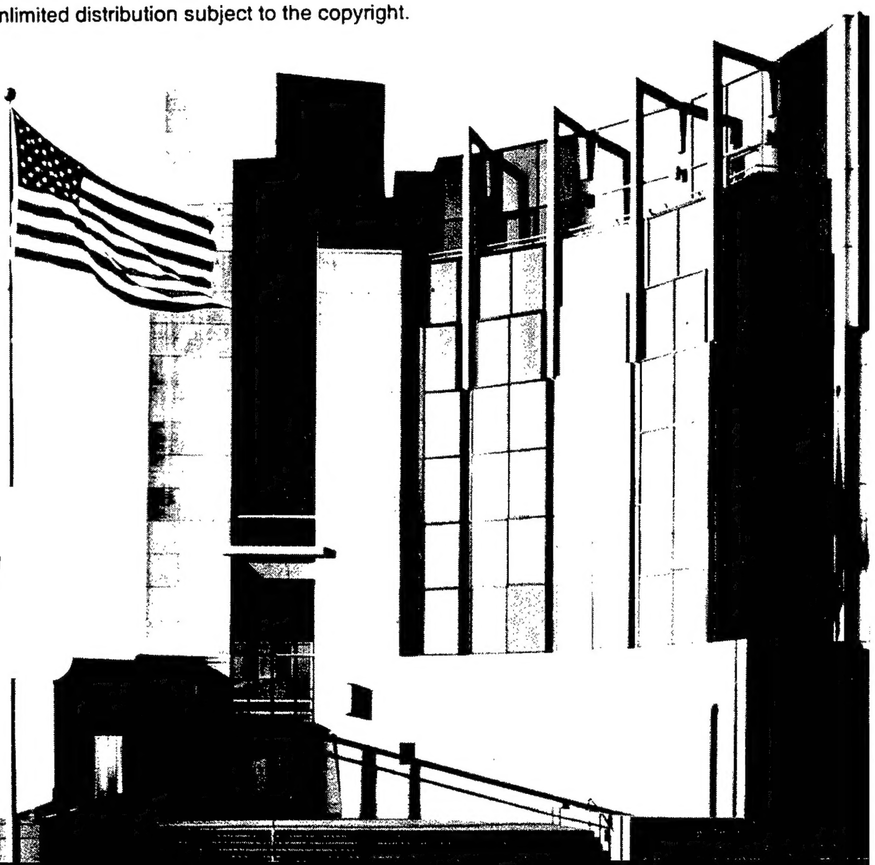
Documenting Software Architectures: Organization of Documentation Package

CMU/SEI-2001-TN-010

Felix Bachmann
Len Bass
Paul Clements
David Garlan
James Ivers
Reed Little
Robert Nord
Judy Stafford

August 2001

Product Line Practice Initiative



Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA 15213-3890

Documenting Software Architectures: Organization of Documentation Package

CMU/SEI-2001-TN-010

Felix Bachmann
Len Bass
Paul Clements
David Garlan
James Ivers
Reed Little
Robert Nord
Judy Stafford

August 2001

Product Line Practice Initiative

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

Norton L. Compton

Norton L. Compton, Lt Col., USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

last printed October 15, 2001 8:19 am / v2.0 / bw4le

Table of Contents

List of Figures

Abstract

This report represents a milestone of a work in progress. That work is a comprehensive handbook on how to produce high-quality documentation for software architectures. The handbook, tentatively entitled *Documenting Software Architectures,*[1] will be published in early 2002 by Addison Wesley Longman as part of the SEI Series on Software Engineering. Since this report is a snapshot of current work, the material described here may change before the handbook is published.

The theme of the report is that documenting an architecture entails documenting the set of relevant views of that architecture, and then completing the picture by documenting information that transcends any single view. The audience for *Documenting Software Architectures* is the community of practicing architects, apprentice architects, and developers who receive architectural documentation.

1. A previous working title was *Software Architecture Documentation in Practice.*

1 Introduction

This report represents a milestone of a work in progress. That work is a comprehensive handbook on how to produce high-quality documentation for software architectures. The handbook, tentatively entitled *Documenting Software Architectures,*[1] will be published in early 2002 by Addison Wesley Longman as part of the SEI Series on Software Engineering. Since this report is a snapshot of current work, the material described here may change before the handbook is published.

Why are we writing one more book on software architecture? After all, there is no shortage of material on the importance of architecture. There is less, but still lots of material on tools for crafting an architecture using styles and patterns. And there is an over-abundance of material on using particular design notations such as the Unified Modeling Language (UML). Yet there is a lack of language-independent guidance about how to capture an architecture in a written form; one that can provide a unified design vision to all the stakeholders on a development project. The current technical note helps to address this need.

The theme of the note is that documenting an architecture entails two essential steps: 1) documenting the set of relevant views of that architecture and then completing the picture by 2) documenting information that transcends any single view. The audience for *Documenting Software Architectures* is the community of practicing architects, apprentice architects, and developers who receive architectural documentation.

A previous report [Bachmann 00] laid out our approach and organization for the complete book and provided full guidance for one of the most commonly used architectural views: the *layer diagram.*

1. A previous working title was *Software Architecture Documentation in Practice.*

2 Software Architectures and Documentation

Software architecture has emerged as an important sub-discipline of software engineering, particularly in the realm of large system development. Architecture gives us intellectual control over a complex system by allowing us to focus on the essential components and their interactions, rather than on extraneous details. Carefully partitioning a whole into parts (with specific relations among the parts) allows groups of people—often groups of groups of people separated by organizational, geographical, and even temporal boundaries—to cooperate productively to solve a much larger problem than they could individually. It's "divide and conquer," followed by "mind your own business," followed by "how do these things work together?" Each part can be built knowing very little about the other parts except that, in the end, these parts must work together to solve the larger problem. A single system is almost inevitably partitioned simultaneously in a number of ways, e.g., different sets of parts or different relations among the parts.

The properties that the system exhibits as it executes are among the most important issues to consider when designing, understanding, or implementing a system's architecture. What the system computes is, of course, one of these issues. But nearly as important are properties (i.e., quality attributes) such as performance, reliability, security, or modifiability. The architecture must be documented to communicate how it achieves those properties.

Before we discuss the forms of architecture documentation, we will discuss the uses of architecture documentation since its uses will determine its forms. Fundamentally, architecture documentation can serve three different functions:

1. *A means of education.* Typically, this means introducing people to the system. The people may be new members of the team, external analysts, or even a new architect.
2. *A vehicle for communication among stakeholders.* A stakeholder is someone who has a vested interest in the architecture. The documentation's use as a communication vehicle will vary according to which stakeholders are communicating, as the following examples show:
   - Downstream designers and implementors use documentation to obtain their "marching orders." The documentation establishes inviolable constraints (plus exploitable freedoms) on downstream development activities.
   - Testers and integrators rely on documentation to specify the correct black-box behavior of the pieces that must fit together.

- Managers use documentation to help them assemble teams by work assignments, organizational structure, planning requirements, project resources, and milestones.
- Designers of other systems use the documentation to define the set of operations provided and required, and the protocols necessary for technical compatibility.
- The architect and requirements engineers representing the customer(s) use documentation as a forum for negotiating and making tradeoffs among competing requirements.
- The architect and component designers also use it to arbitrate resource contention and to establish performance and other kinds of run-time resource consumption budgets.
- Product line managers rely on it to determine whether a potential new member of a product family is in or out of scope; and if out, by how much.
- Technical mangers use documentation for conformance checking and for assuring that implementations have, in fact, been faithful to architectural prescriptions.
- System maintainers use documentation architecture as the starting point for maintenance and, eventually, upgrading activities.

3. *A basis for system analysis.* To support analysis, the documentation must provide the appropriate information for the particular activity being performed. For example,
   - For performance engineers, the documentation must provide the formal model that drives analytical tools such as rate-monotonic real-time schedulability analysis, simulations and simulation generators, theorem provers, and model checking verifiers. These tools require information about resource consumption, scheduling policies, dependencies, and so forth.
   - For those interested in the design's ability to meet system quality objectives, the documentation serves as the input for architectural evaluation methods. It must contain the information necessary to evaluate attributes such as security, performance, usability, availability, and modifiability.

Architecture documentation must balance these varied purposes. It should be sufficiently abstract that it is quickly understood by new employees. It should be sufficiently detailed so that it serves as a blueprint for construction. At the same time, it should have enough information so that it can serve as a basis for analysis.

Furthermore, architecture documentation is both prescriptive and descriptive. That is, it prescribes what should be true by placing constraints on decisions that are about to be made, and it describes what is true by recounting decisions that already have been made. However, the best architectural documentation for performance analysis may well differ from the best documentation for system integrators. Both of these will differ from the documentation that a new-hire receives. The documentation planning and review process must support all relevant needs.

3 Architectural Views

Perhaps the most important concept associated with software architecture documentation is the *view.* A software architecture is a complex entity that cannot be described in a simple one-dimensional fashion. The analogy with a building architecture, if not taken too far, proves illuminating. There is no single rendition of a building architecture. Instead, there are room layouts, elevations, electrical diagrams, plumbing diagrams, HVAC system diagrams, traffic patterns, sunlight and passive solar views, security system plans, and many others. Which of these views *is* the architecture? None of them. Which views *convey* the architecture? All of them.

A view, then, represents a set of system elements and their relationships. A view documents a particular aspect of the system's architecture while intentionally suppressing others. Different views will highlight different system elements and/or relationships.

Several authors have prescribed specific views that practitioners should employ to document their software architectures. In particular, Philippe Kruchten of the Rational Corporation wrote a very influential paper describing four main views of software architecture that can be used to great advantage in system-building, plus a distinguished fifth view that ties the other four together—the so-called "4+1" approach to architecture [Kruchten 95]:

- The *logical view* primarily supports behavioral requirements—the services the system should provide to end users. Designers decompose the system into a set of key abstractions taken mainly from the problem domain. These abstractions are objects or object classes that exploit the principles of abstraction, encapsulation, and inheritance. In addition to aiding functional analysis, decomposition identifies mechanisms and design elements that are common across the system.
- The *process view* addresses concurrency and distribution, system integrity, and fault tolerance. It also specifies which thread of control executes each operation of each class identified in the logical view. The process view can be seen as a set of independently executing logical networks of communicating programs ("processes") that are distributed across a set of hardware resources, which in turn are connected by a bus, local area network (LAN), or wide area network (WAN).
- The *development view* focuses on the organization of the modules in the software development environment. The units of this view are small chunks of software, usually program libraries or subsystems. The development view supports allocating requirements and work

to teams. It also supports cost evaluation, planning, monitoring project progress, and reasoning about software reuse, portability, and security.

- The *physical view* presents the system's requirements such as availability, reliability (fault-tolerance), performance (throughput), and scalability. This view maps the various elements identified in the logical, process, and development views—networks, processes, tasks, and objects—onto the processing nodes.

Finally, Kruchten prescribes using a small subset of important scenarios—instances of use cases—to show that the elements of the four views work together seamlessly. This is the "plus one" view, redundant with the others but serving a distinct purpose. 4+1 has since been embraced as a foundation piece of the Rational Unified Process [Kruchten 98].

While those are indeed useful views in general, they are not useful for every system, and do not constitute a closed set. The point is that a view is a powerful mechanism for separating concerns and communicating the architecture to a variety of stakeholders. This leads to a fundamental principle of software architecture documentation:

Documenting an architecture is a matter of documenting the relevant views and their relationships, and adding documentation that applies to more than one view.

This principle is illustrated in Figure 1:

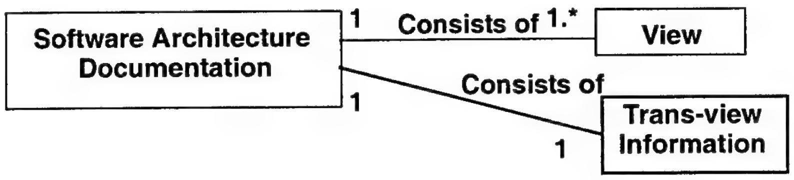


Figure 1: Entity-Relation Diagram of Software Architecture Documentation[1]

1. In this report, we use entity-relation diagrams like the one above to show how concepts fit together. The relations are presented as labels on lines that connect boxes or entities. The diagrams are read top to bottom or left to right. Numbers mark each end of the line and indicate the entities of the relation. An integer followed by "..*" means at least the value of that integer, whereas a solitary "*" means a zero or more. For example, Figure 1 should be read as, "Software architecture documentation consists of one or more views and one trans-view information set."

Which are the relevant views? It depends on your goals. Architecture documentation can serve many purposes: a mission statement for implementors, a basis for analysis, the specification for automatic code generation, the starting point for system understanding and asset recovery, or the blueprint for project planning. Different views support different goals and uses, and so another tenet of documentation is that what you write down depends on what you expect to do. This is fundamentally why we do not advocate a particular view or collection of views.

Sometimes, for example, it is convenient and useful to show information that is native to more than one view. In this case, you may opt for a *hybrid view*. It contains elements and relationships that come from two or more views. For example, a deployment view is such a hybrid. It shows how processes (normally shown in a process view) are deployed onto the hardware (normally shown in a physical view). Deployment views are invaluable for reasoning about performance. The point is, carefully chosen hybrid views often provide the most insight of all architectural information. At the same time, "accidental" hybrids, views that unintentionally conflate unrelated information, can be one of the greatest sources of architectural confusion.

Since all views describe system elements and the relationships among them, an architect can employ a common organizational scheme to document them. That common organizational scheme should contain the following:

- A *primary presentation* that shows elements and relationships. The primary presentation is usually graphical. If so, this presentation must be accompanied by a key that explains or points to an explanation of the notation used in the presentation. If the primary presentation is textual instead of graphical, it still carries the obligation to present a terse summary of the most important information in the view. If that text is presented according to certain stylistic rules, the rules should be stated as the analog to the graphical notation key.
- A suite of *supporting documentation* that explains and elaborates the information in the primary presentation.

Every view, then, consists of a primary presentation, usually graphical, and supporting documentation that explains and elaborates the pictures. To underscore this point, we call the graphical portion of the view an *architectural cartoon*. We use the definition from the world of fine art, where a cartoon is a preliminary sketch of the final work. It reminds us that the picture, while getting most of the attention, is not the complete view description but only a sketch of it. In fact, it is merely an introduction to, or a quick summary of, the view that is completely described by the supporting documentation.

The primary presentation should contain the information you wish to convey about the system (in the vocabulary of that view) first. It should certainly include the primary elements and relations of the view, but under some circumstances it might not include all of them. For example, you may wish to show the elements and relations that come into play during normal operation, but relegate error-handling or exceptional processing to the supporting documentation. What information you include in the primary presentation may also depend upon what notation you use, and how conveniently it conveys various kinds of information. A richer notation will tend to enable richer primary presentations. Figure 2 summarizes:

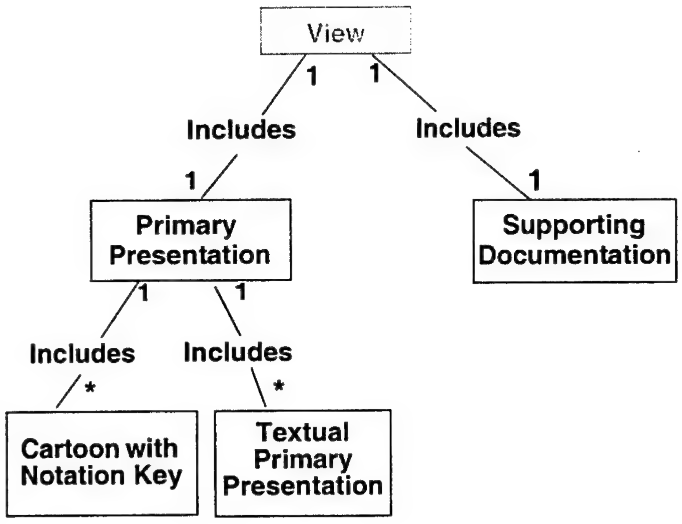


Figure 2: View Is Documented By a Primary Presentation and Supporting Documentation

As shown in Figure 3 the supporting information documents the view in depth.

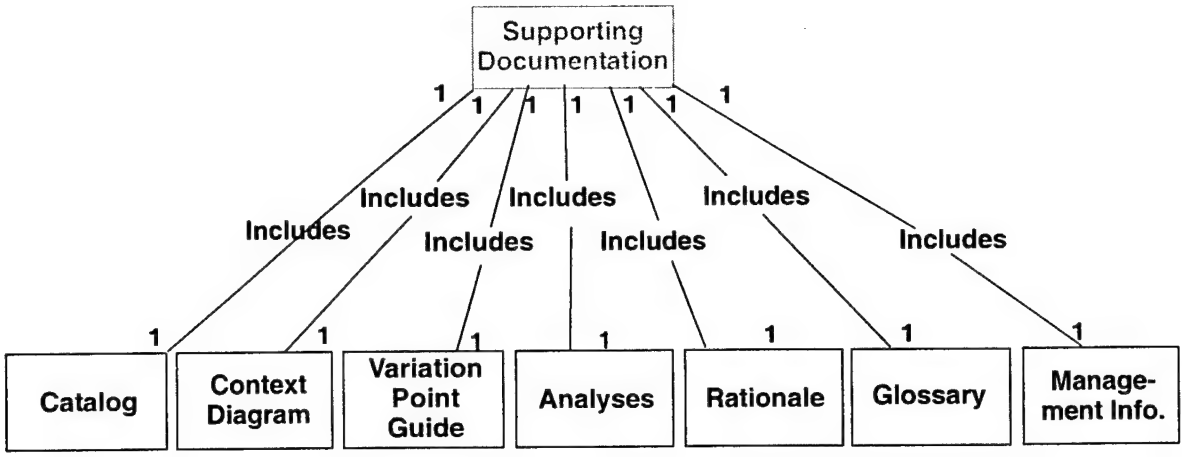


Figure 3: Suite of Supporting Documentation

The suite of supporting information includes a *catalog* that defines the elements and relationships of the view, including at a minimum those shown in the primary presentation, and perhaps others (see the discussion of descriptive completeness in Section 4). For instance, if a diagram shows elements A, B, and C, then there had better be documentation that explains in sufficient detail what A, B, and C are, and their purposes or roles they play, rendered in the vocabulary of the view. This explanation should include the interface(s) of the elements and might additionally include a list of properties, and/or a behavioral description. In addition, if there are elements or relations relevant to the view that were omitted from the primary presentation, the catalog can present them. Figure 4 illustrates the contents of a view's catalog:

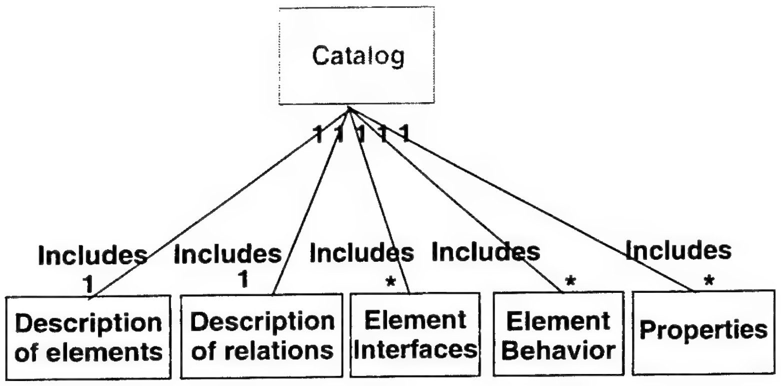


Figure 4: Contents of a Catalog

The catalog presents information that defines the elements and relations of the view. In addition to the catalog, the supporting documentation should include

- a *context diagram* that delineates the boundary between the entity whose architecture is being documented in that particular view and its environment
- *a guide showing how to exercise any variation points* that are a part of the architecture shown in this view
- *results of analyses* that have been conducted, such the results of performance or security analysis, or a list of changes required by a particular modification
- *rationale* for why the design decisions reflected in the view were made along with a list of rejected alternatives and why they were rejected
- *glossary* of terms used

- *project management related information*. The precise contents of this section will vary according to the standard practices of each organization, but this is the place where information such as authorship, configuration control data, and change histories are recorded.
- *a view template*. This template lets an architect convey a view to the relevant stakeholders in a standard way. Figure 5 captures this.

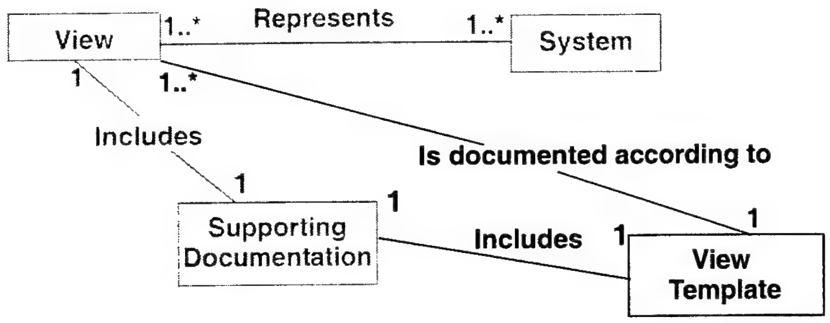


Figure 5: View Template

Every view is documented according to a view template, so that stakeholders can understand the documentation's rules and organization. More than one view might be documented according to the same view template, in which case it should be incorporated into a view's supporting documentation by reference.

4 Chunking Information: View Packets, Refinement, and Descriptive Completeness

4.1 View Packets

Views of large software systems can contain hundreds or even thousands of elements. Showing these in a single presentation along with their relations can result in a blizzard of information that (a) is indecipherable and (b) contains far too much data for those stakeholders who are only concerned with a certain part of the system. We need a way to present a view's information in digestible "chunks." We call these chunks *view packets*.

Each view packet shows a fragment of the system. Think of it as the smallest bundle of documentation that you would give to a development team, subcontractor, or other stakeholder. Figure 6 shows that the documentation for a view comprises a set of view packets.

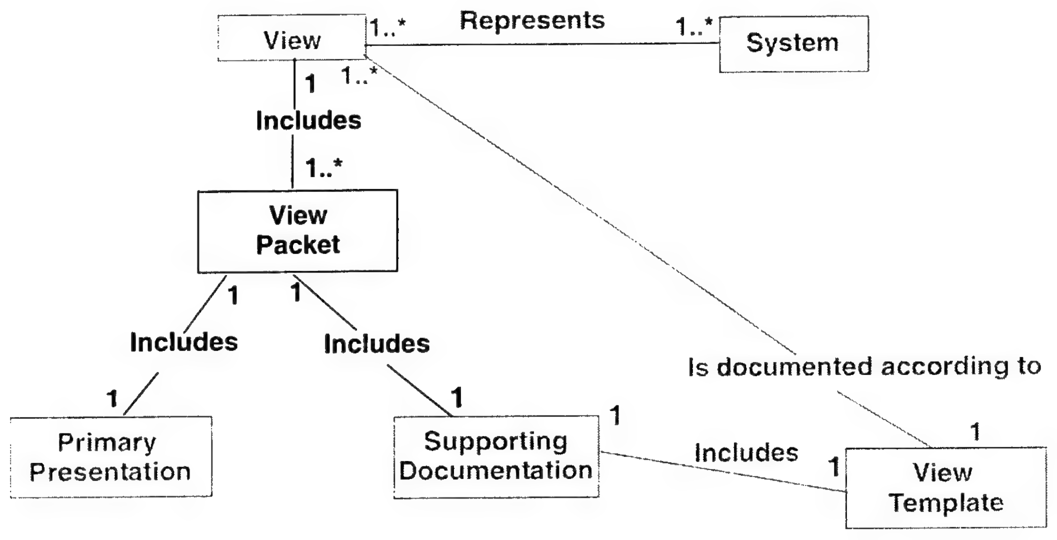


Figure 6: A View Consists of One or More View Packets

Each view packet, in turn, contains a primary presentation and its supporting documentation. However, some information in the supporting documentation applies to all view packets. The glossary, for instance, will apply to the entire view, and the view template certainly does. By contrast, the catalog and variation guide are more likely to apply to specific view packets.

Some kinds of information might apply to both the entire view and a particular view packet. For example, some rationale might apply to the portion of a design captured by a view packet as well as to the entire view. We also would expect management information (such as configuration control and authorship information) to be included with each view packet, as well as with the entire view.

We update Figure 3 to show that each view (in addition to consisting of a collection of view packets) has a supporting documentation package containing information that applies to the entire view.

At the same time, each view packet also has a supporting documentation package. The architect has broad latitude about organizing information to best serve stakeholders. If there is a piece of rationale that applies to the portion of a design shown in a view packet, then perhaps it should be documented alongside that view packet. On the other hand, perhaps the architect chooses to record all rationale in one place. In this case, the piece of rationale is most conveniently recorded in the supporting documentation for the entire view. Remember that a view packet is a bundle of cohesive documentation that you would give to a stakeholder. Use this to determine what supporting documentation the stakeholder(s) who "own" a view packet will want. See Figure 7.

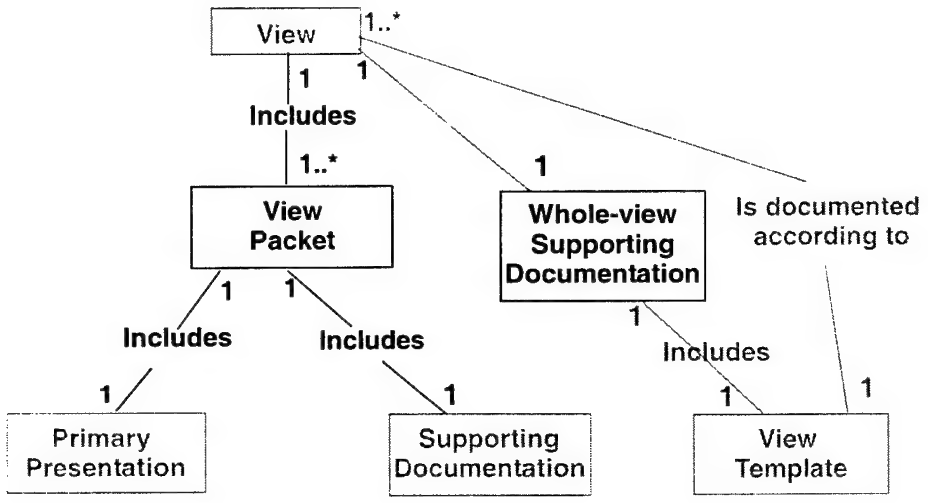


Figure 7: View with Whole-View Supporting Documentation

View packets that constitute a view are related to each other in either of the following ways:

- The view packets are siblings of each other, meaning that they document different parts of the same system. Think of these view packets as forming a mosaic of the whole view, as if each was a photograph taken by a camera that panned and tilted across the entire view.
- Some view packets are children of another, meaning that they document the same part of the system but in greater detail. Think of these view packets as coming into focus when our hypothetical camera zooms in on a part of the system.

View packets allow the architect to document a view (and a reader to understand a view) in

- depth-first order (that is, choosing an element, documenting its sub-structure, choosing a sub-element, documenting its sub-structure, etc.)
- breadth-first order (for all elements, document their sub-structures then for all of those elements, document their sub-structures, etc.)
- some combination of the two based on what the architect knows at the time

View packets give the architect the flexibility to let a view be used by different stakeholders with different concerns. If an overview of the system needs to be conveyed, then view packets showing coarse-grained information can be used, and can do the job in a small number of pages. On the other hand, if the goal is reasoning about the achievement of a particular quality such as performance, then details of the elements and relations will need to be conveyed. In this case, view packets that show finer-grained information should be used.

4.2 Refinement

We say that the view packets that represent a zoom-in operation are *refinements* of their parent. Architects use *refinement*, the gradual disclosure of information across a series of descriptions, to represent the information in a view.

A *decomposition refinement* (or simply decomposition) elaborates a single element to reveal its internal structure, and then recursively refines each member of that internal structure. The text-based analogy of this is the outline, where Roman-numeral sections are de-composed into capital-letter sections which are de-composed into Arabic-numeral sections, which are de-composed into small-letter sections, and so forth. Figure 8a is a cartoon showing three elements. Figure 8b shows that element B consists of four elements. In this cartoon, element B1 has the responsibility of handling element B's interactions with other elements.

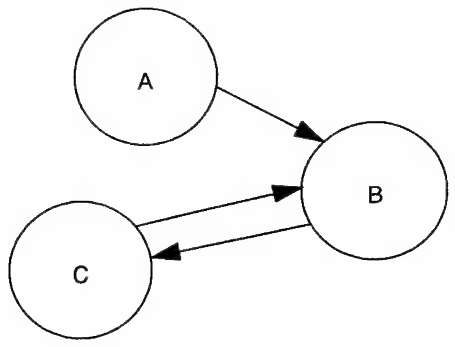


Figure 8a: Three Element System

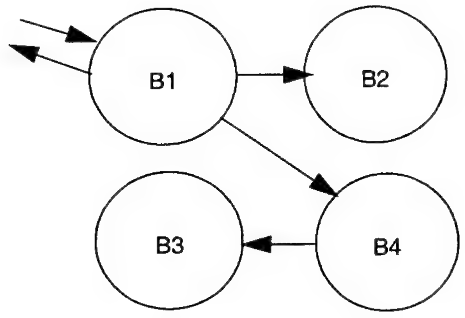


Figure 8b: A Four Element System Connected to the Outside

Using decomposition refinements carries an obligation to maintain consistency with respect to the relation(s) native to that view. For example, suppose the relation shown in Figure 8a is "sends data to." Since element B is shown as receiving as well as sending data, then the refinement of B in Figure 8b must show where data can enter and leave B—in this case, via B1.

Another kind of refinement is an *implementation refinement.* This shows the same system (or portion of the system) in which many or all of the elements and relations are replaced by new (typically more implementation-specific) ones. For example, imagine two views of a publish-subscribe system. In one view, components are connected by a single event bus. In the refined view, the bus is replaced by a dispatcher that receives calls from the components and makes event announcements. Note that replacing the connector forces us to change the interfaces of the components—hence we have an implementation refinement. If an implementation refinement introduces vocabulary from a view that did not appear in the original view packet, then this is a special case of trans-view documentation—where learning about a system takes a reader from one view to another.

4.3 Descriptive Completeness

Related to refinement is the concept of *descriptive completeness*, which tells how view packets are related to each other.

Figure 9 shows an architectural cartoon for some imaginary system. It tells us that element A is related to element B in some way (the cartoon does not disclose how), B is related to C, and C is related to B.

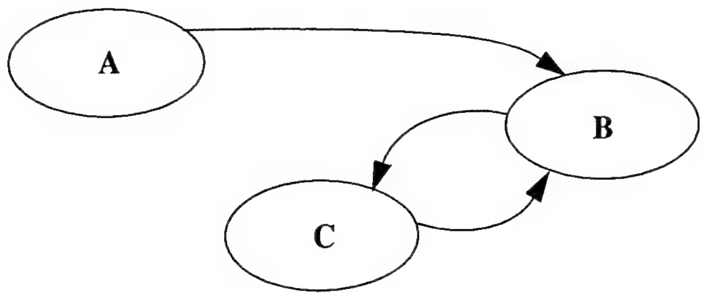


Figure 9: Redrawn Three Element System

What can we conclude about whether or not A and C are related?

There are two possible answers. The first one is straightforward: "A and C are not related, because the diagram shows no arrow between A and C." The second a bit more complicated: "This diagram reveals no relationship between A and C, but it is possible that this information was considered too detailed or tangential for the view packet in which this cartoon appears. Therefore, we cannot answer the question at this time. Another view packet may subsequently reveal that A and C share this relation."[1]

Either answer is acceptable, as each represents a different strategy for documentation. The first strategy says that the view packets are written with *descriptive completeness*. This strategy tends to be used for packets that convey instructions or constraints to downstream designers or implementors. For instance, one common architectural view (the layered view) shows implementors what other elements they are allowed to use (and, by extension, what elements they are prohibited from using) when coding their work assignments. If we gave the coder of

element A the cartoon in Figure 9, we would want him or her to interpret the absence of an arrow between A and C as carrying meaning—namely, a prohibition from using element C.

1. There is actually a third case. If we happen to know that relation (whatever it might be) is transitive, then we could deduce that it holds between A and C since it holds between A and B and between B and C. That case is not relevant for discussion at hand, however.

The second strategy tends to be used for view packets that convey broad understanding. Suppose we want to picture a system's data flow. In that case, we might not wish to show total data flow, but only the data flow in the nominal, usual, or high-frequency cases. We might defer data flow to another view packet when the system is doing, say, error detection and recovery. Suppose Figure 9 shows that nominal case. Figure 10 might show the error case. A programmer may eventually want to see both diagrams, but not at once.

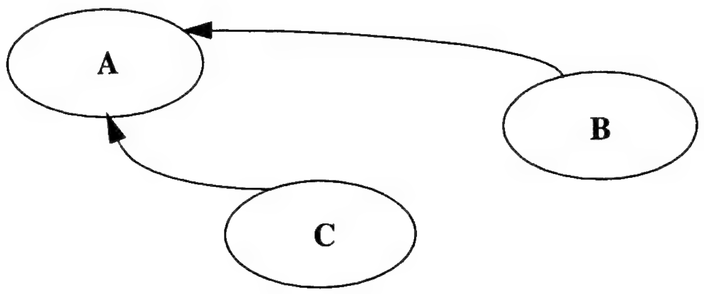


Figure 10: Three Element System Showing Alternative Relationship

Under this interpretation, Figure 10 does not contradict Figure 9 but augments it, whereas under the assumption of completeness, Figure 9 and Figure 10 are contradictory. Both cannot document the same system.

Up to this point, we've discussed these strategies in terms of relationships among elements, but we could also ask an element-related question. Suppose Figure 9 purports to show an entire system, or a specific portion of it. Can we presume that A, B, and C are the only elements in (that portion of) the system? That is, is every piece of software either in A or B or C? The same two strategies apply. The first tells us "Yes, you can presume that with impunity. All software within the scope of this view packet is shown in this view packet." The second says "We don't know yet. Perhaps in a refinement or augmentation of this view, another element will be shown." If an error-logging element comes into play during error detection, a diagram like the one in Figure 11 might apply.

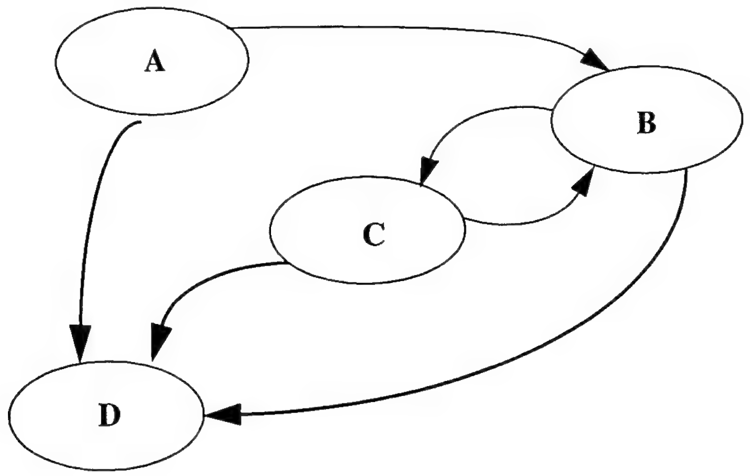


Figure 11: A Supplement to the Cartoon of Figure 9, Showing an Additional Component

Again, both strategies are correct and have their place. A fundamental principle of graphical documentation is to explain your notation. The issue of descriptive completeness is a special case of that. You simply need to specify which of the two strategies your documents follow. As we suggested, some documents may follow one while others may follow the other. That is not a problem, as long as the reader is informed. So, for example, if you adopt the completeness strategy you might include something like the following in your documentation, perhaps as part of the notation key:

> **If no relationship is shown between two components, then that relationship {does not, is not allowed to} exist between those two components.**

Or this

> **The elements shown in this diagram account for all the software in the system.**

If you adopt the non-completeness strategy, then you might have explanations like this as your key:

> **Subsequent refinements or other renditions of this view (give pointer) may show relationships among elements that are not shown here.**

Earlier we dealt with the issue of refinement. The descriptive completeness issue is related. If your view packets convey descriptive completeness, then this conveys obligations on refinements. If a view packet shows no relationship (of whatever kind) between, say, elements A and C then no refinement of that view packet is allowed to subsequently show that relationship between A and C. If a view packet shows a set of elements, then no new elements may be introduced later.

Descriptive completeness makes consistency checking among view packets much more straightforward, but at the cost of making the cartoons and their explanation more cluttered and arguably harder to understand. As in many issues of architecture, this one brings with it an inherent tradeoff.

5 Trans-view Information

Trans-view information, which may be the subject of a future technical note, consists of four primary categories:

1. **Mapping between views.** Views, like the perceptions of blind men groping the elephant, are all quite different from each other. In the end, however, they describe the same entity. They are related in ways that are often complex, but which always provide additional insights about the architecture as a whole. Providing relationships among the views is an important aspect of the complete documentation package.
2. **Rationale.** Why an architect made decisions constitutes a crucial but, unfortunately, often overlooked aspect of documentation. While rationale certainly applies to "architecturally local" decisions (such as those within a single view), there is also a rationale that applies across views as well.
3. **Constraints.** No architect has a free hand. It is important to document the constraints that apply to the architecture. Constraints are often subtle, and always far exceed the rules for behavior captured in a requirements specification. They often include real world exigencies such as the reliability of vendors, and organizational factors such as the availability (or non-availability) of qualified staff or other resources.
4. **View catalog.** This piece of documentation is a stakeholder's introduction to the entire package. It explains what documentation is available and where it may be found. It serves as the overall reader's guide to the entire suite.

6 Epilogue

Documenting a software architecture means documenting its relevant views and then recording trans-view information. A view is a representation of a set of system elements and relationships among them. Views are documented (according to a view template) by a set of view packets, which consists of a primary presentation and its supporting documentation. View packets allow information to be presented in digestible chunks. In this regard, view packets are rather like snapshots of a system taken with a camera that tilts, pans, and zooms in and out. Each view includes a view template that outlines the documentation organization and content for that view. Trans-view information consists of a mapping between views, rationale, constraints on the architecture, and a view catalog.

These rules and principles provide a basis for a documentation package that will help a software architecture live up to its potential uses as a vehicle for education, communication, and analysis.

Of course, a practitioner will need more detailed guidance about several aspects of the task before such a package can be produced. Future work will include specifics about trans-view information, view templates, choosing views, and the properties and uses of a wide variety of views.

7 References

[Bachmann 00]	Bachmann, F; Bass, L; Carriere, J; Clements, P; Garlan, D; Ivers, J; Little, R.; & Nord, R. *Software Architecture Documentation in Practice: Documenting Architectural Layers* (CMU/SEI-2000-SR-004). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. Available WWW: <http://www.sei.cmu.edu/publications/documents/00.reports/00sr004.html> (2000).
[Kruchten 95]	Kruchten, P. "The 4+1 View Model of Architecture." *IEEE Software*, Vol 12, No. 6, November 1995.
[Kruchten 98]	Kruchten, P. *The Rational Unified Process: An Introduction.* Reading, MA: Addison-Wesley, 1998.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. **AGENCY USE ONLY** (leave blank)	2. **REPORT DATE** August 2001	3. **REPORT TYPE AND DATES COVERED** Final
4. **TITLE AND SUBTITLE** Documenting Software Architectures: Organization of Documentation Package		5. **FUNDING NUMBERS** F19628-00-C-0003
6. **AUTHOR(S)** Felix Bachmann, Len Bass, Paul Clements, David Garlan, James Ivers Reed Little, Robert Nord, Judy Stafford		
7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		8. **PERFORMING ORGANIZATION REPORT NUMBER** CMU/SEI-2001-TN-010
9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116		10. **SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. **SUPPLEMENTARY NOTES**		
12.a **DISTRIBUTION/AVAILABILITY STATEMENT** Unclassified/Unlimited, DTIC, NTIS		12.b **DISTRIBUTION CODE**

13. **ABSTRACT** (maximum 200 words)

This report represents a milestone of a work in progress. That work is a comprehensive handbook on how to produce high-quality documentation for software architectures. The handbook, tentatively entitled *Documenting Software Architectures,* will be published in early 2002 by Addison Wesley Longman as part of the SEI Series on Software Engineering. Since this report is a snapshot of current work, the material described here may change before the handbook is published.

The theme of the report is that documenting an architecture entails documenting the set of relevant views of that architecture, and then completing the picture by documenting information that transcends any single view. The audience for *Documenting Software Architectures* is the community of practicing architects, apprentice architects, and developers who receive architectural documentation.

14. **SUBJECT TERMS** software architecture documentation, documentation views, templates, documentation users			15. **NUMBER OF PAGES** 33 16. **PRICE CODE**
17. **SECURITY CLASSIFICATION OF REPORT** UNCLASSIFIED	18. **SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	19. **SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	20. **LIMITATION OF ABSTRACT** UL